Abstract
(DE 4206567)

The method includes the step of generating a number of sequences of programme parts pseudo-stochastically, and evaluating them with reference to the processing or running time. A first sequence, giving the shortest time, is selected and then this first sequence is processed further and, pseudo-stochastically, two programme parts are exchanged and examined to see if the time has been shortened. When this is so, the changed sequence is further processed and compared with the processing of the original sequence. This procedure is continued until the most efficient conjunction of the parts has been found.

USE/ADVANTAGE - Suitable for data processing systems operating to cache principle. Programme parts can be bound into existing programme to result in minimum running time for new programme.

(19) BUNDESREPUBLIK DEUTSCHLAND

DEUTSCHES PATENTAMT

(12) **Patentschrift**

(10) **DE 42 06 567 C 1**

(51) Int. Cl.[5]:
G 06 F 9/45
G 06 F 12/08

(21) Aktenzeichen: P 42 06 567.4-53
(22) Anmeldetag: 2. 3. 92
(43) Offenlegungstag: —
(45) Veröffentlichungstag der Patenterteilung: 27. 5. 93

DE 42 06 567 C1

(73) Patentinhaber:
Siemens AG, 8000 München, DE

(72) Erfinder:
Lindmeier, Horst, Dipl.-Ing.; Hafer, Christian, Dipl.-Ing., 8000 München, DE; Plankl, Josef, Dipl.-Inform., 8011 Siegertsbrunn, DE; Gösmann, Klaus, Dipl.-Phys.; Westerholz, Karl, Dipl.-Ing., 8000 München, DE

(56) Für die Beurteilung der Patentfähigkeit in Betracht gezogene Druckschriften:
DE 41 08 309 A1

(54) Verfahren zum Binden von Programmteilen zu einem Programm

(57) Die in einem Programm enthaltenen Programmodule oder Programmteile werden so zu einem Programm gebunden, daß dessen Laufzeit auf einen nach dem Cache-Prinzip arbeitenden Rechner minimal ist. Dabei wird pseudozufällig eine Mehrzahl von Reihenfolgen der Programmteile erzeugt und diese Reihenfolgen bezüglich der Laufzeit bewertet. Es wird die Reihenfolge der Programmfolge als aktuelle Reihenfolge ausgewählt, die das beste Bewertungsergebnis, also die geringste Laufzeit hat. Mit dieser aktuellen Reihenfolge der Programmteile wird weiter gearbeitet und dazu jeweils pseudozufällig zwei Programmteile der aktuellen Reihenfolge ausgewählt und vertauscht und dann wiederum festgestellt, ob die durch Vertauschung entstandene neue Reihenfolge der Programmteile zu einem besseren Bewertungsergebnis führt als die aktuelle Reihenfolge von Programmteilen. Ist dies der Fall, dann wird die neue Reihenfolge zur aktuellen Reihenfolge, sonst wird mit der bisherigen aktuellen Reihenfolge durch erneute Vertauschung von zwei Programmteilen weitergefahren. Dieser Schritt wird solange durchgeführt, bis ein vorgegebenes Abbruchkriterium erfüllt ist, der z. B. in einer vorgegebenen Anzahl von Vertauschungen liegen kann.

DE 42 06 567 C1

Beschreibung

Die Anmeldung betrifft ein Verfahren zum Binden von Programmteilen zu einem Programm, das von einem nach dem Cache-Prinzip arbeitenden Rechner mit minimaler Laufzeit abgearbeitet werden kann.

Cache-Speicher oder Cache gehören zu den Standard-Speicher-Hierarchien, die die Architektur eines Rechners mitbestimmen. Ihre Funktion ist die eines Pufferspeichers zwischen einem Hauptspeicher und einem Prozessor. Der Cache als Pufferspeicher soll hierbei den schnellen Prozessor möglichst von dem langsamen Hauptspeicher entkoppeln. Der Cache besteht z. B. aus schnellen S-RAMS, die genau so schnell sind wie der Prozessor. Aufgrund ihres hohen Preises sind die Caches aber wesentlich kleiner als der Hauptspeicher. Speicherhierarchien mit Cache sind bekannt und werden darum in ihrer Funktion nicht weiter erläutert.

Die Leistungsfähigkeit von Cache-Architekturen beruht auf dem Prinzip der Programm-Lokalität, ohne die eine Speicher-Hierarchie aus Cache und Hauptspeicher keinen Leistungsgewinn bringen würde. Bei der Programm-Lokalität wird zwischen zeitlicher und räumlicher Lokalität unterschieden. Zeitliche Lokalität besagt, daß, wenn ein Datum adressiert wurde, es wahrscheinlich ist, daß es bald wieder adressiert wird. Räumliche Lokalität besagt, daß, wenn ein Datum adressiert wurde, es wahrscheinlich ist, daß weitere Daten in seiner Nähe adressiert werden.

Um Daten zwischen Hauptspeicher und Cache zu transportieren, werden der Hauptspeicher und der Cache in sog. Cache-Blöcke unterteilt, die typisch 4 bis 128 Bytes enthalten können. Dabei können diese Daten sowohl Programmteile sein als auch Datenworte, die vom Programm bearbeitet werden.

Es gibt Design-Möglichkeiten des Caches, die für die Leistungsfähigkeit des gesamten Rechnersystems entscheidend sind. Weiterhin wird die Leistung des Cache aber auch von der verwendeten Software beeinflußt. Ein Cache hat seine maximale Leistung erreicht, wenn bei jedem Zugriff auf den Cache das angeforderte Datum im Cache vorhanden ist. Dieser Fall wird als Cache-Hit bezeichnet. Der Idealfall tritt jedoch immer nur kurzzeitig auf und wird durch Fehlzugriffe, sog. Cache-Misses, immer wieder unterbrochen, bei denen sich der Prozessor sich die benötigten Daten aus dem langsameren Hauptspeicher holen muß.

Die Wirksamkeit eines Cache ist somit einerseits von der Fehlzugriffsrate (Miss-Rate) als auch von der Bearbeitungszeit für einen Fehlzugriff abhängig. Die Bearbeitungszeit selbst ist von der Reaktionszeit des Hauptspeichers, der Cache-Block-Größe und der Busbandbreite abhängig. Die Fehlzugriffsrate ist ebenfalls von der Cache-Blockgröße abhängig. Bei großen Cache-Blöcken sinkt die Miss-Rate aufgrund der räumlichen Programm-Lokalität, jedoch steigt andererseits die Bearbeitungszeit für einen Fehlzugriff. Das Optimum der Cache-Blockgröße ist von der gewählten Cache-Architektur wie auch von der Anwendung abhängig. Die Cache-Blöcke sind jedoch nicht die Ursache für einen Fehlzugriff. Hierfür ist die Programmlast, der Compiler und das Betriebssystem verantwortlich. Dabei muß gesagt werden, daß ein bestimmter Anteil von Fehlzugriffen unvermeidbar ist.

Eine Art des Fehlzugriffes ist der sog. Verdrängungs-Miss oder Kollisions-Miss. Voraussetzung für einen Verdrängungs-Miss ist, daß der Inhalt des adressierten Speichers sich bereits einmal im Cache befand. Dieser wurde jedoch von später adressierten Speicherinhalten überschrieben, so daß eine abermalige Adressierung zum Miss führt. Die Anzahl solcher Verdrängungs-Misses ist u. a. abhängig von der Cache-Größe und der Reihenfolge der einzelnen Programmteile, die zu dem Programm gebunden werden. Ein Programmteil kann dabei z. B. ein Programmodul, eine Prozedur oder eine sonstige Befehlssequenz sein, z. B. auch ein Basisblock, also eine lineare Befehlsfolge. Die Reihenfolge solcher Programmteile hat einen Einfluß auf die Anzahl der Verdrängungs-Misses. Es gibt Reihenfolgen von Programmteilen, bei denen die Anzahl der Verdrängungs-Misses besonders klein ist.

Das der Erfindung zugrundeliegende Problem besteht darin, ein Verfahren anzugeben, mit dem eine Reihenfolge von zu einem Programm zu bindenden Programmteilen ermittelt werden kann, bei der die Laufzeit minimal ist. Dieses Problem wird gemäß den Merkmalen des Anspruchs 1 gelöst.

Es wird somit eine gezielte Suche im Permutationsraum der möglichen Reihenfolgen der Programmteile durchgeführt. Vorausgesetzt wird dabei, daß eine Bewertungsmöglichkeit für die Güte bezüglich der Laufzeit für eine Reihenfolge der Programmteile existiert. Eine solche Bewertungszahl stellt z. B. die Anzahl der benötigten Zyklen eines Programmlaufs dar. Diese Zahl kann z. B. aus Trace-Daten mit Hilfe eines Cache-Simulators gewonnen werden. Als eine weitere Möglichkeit zur Bewertung der Reihenfolge der Programmteile bezüglich der Laufzeit kann die Bewertung durch eine Kostenfunktion erfolgen, die den einzelnen Reihenfolgen der Programmteile Kosten zuordnet.

Die Bewertung entsprechend der Kostenfunktion geht aus von der Durchführung eines Trace-Verfahrens des zu untersuchenden Programms, das als Trace-Ergebnis Adreßreferenzen oder Adreßaufrufe enthält, die sich auf Code-Bereiche oder Datenbereiche beziehen. Das Trace-Verfahren wird ein einziges Mal durchgeführt und kann dann immer verwendet werden. Anschließend wird aus dem Trace-Ergebnis ermittelt, wie oft sich je zwei Programmteile höchstens gegenseitig verdrängen können, falls sie sich im Cache überlappen. Auch diese Auswertung muß nur einmal erfolgen, da die ermittelten Werte unabhängig von der Cache-Größe und der Reihenfolge der Programmteile des Programms sind. Für die Vorhersage der Verdrängungs-Misses einer Reihenfolge von Programmteilen bei fester Cache-Größe wird für jedes aus dem Trace-Ergebnis ermittelte Paar von Programmteilen untersucht, wie groß gemessen in Cache-Blöcken die Überlappung der beteiligten Programmteile ist. Anschließend für jedes Paar von Programmteilen das Produkt zwischen der ermittelten Anzahl der kollidierenden Cache-Blöcke und der maximalen möglichen Anzahl von Verdrängungen gebildet. Die Summe über alle diese Produkte liefert dann ein Maß für die Anzahl der auftretenden Verdrängungs-Misses bei gegebener Cache-Größe und gegebener Reihenfolge der Programmteile eines Programms.

Die Bestimmung der Anzahl der Verdrängungen kann z. B. mit Hilfe eines LRU-Stackspeichers (LRU = Least Recently Used) erfolgen.

Mit dem erfindungsgemäßen Verfahren lassen sich mit geringem Rechenzeitaufwand gute Konfigurationen der Programmteile-Reihenfolgen finden, deren Kosten nur geringfügig höher sind als die der optimalen Konfiguration. Um noch bessere Konfigurationen zu finden, ist auszuschließen, daß sich die Trajektorie im Permutationsraum in einem lokalen Minimum der Kosten verfängt. Um dies zu vermeiden, müssen Programmteile-Vertauschungen mit einer gewissen Wahrscheinlichkeit auch dann zugelassen werden, wenn sich höhere Kosten ergeben. Das kann so durchgeführt werden, daß das Verfahren durch eine Bedingung ergänzt wird, bei der die neue Reihenfolge von Programmteilen auch dann weiter untersucht wird, wenn eine Kostenverschlechterung eintritt, die kleiner ist als eine Zahl, die z. B. von einem Pseudozufallsgenerator erzeugt worden ist.

Andere Weiterbildungen der Erfindung ergeben sich aus den Unteransprüchen.

Anhand von Ausführungsbeispielen, die in den Figuren dargestellt sind, wird die Erfindung weiter erläutert. Es zeigt

Fig. 1 die prinzipielle Darstellung mehrerer Reihenfolgen der Programmteile bei einem Programm zur Erläuterung des Verfahrens,

Fig. 2 eine weitere Reihenfolge der Programmteile des Programms zur Erläuterung des Verfahrens,

Fig. 3 eine prinzipielle Darstellung eines Stack-Speichers, wenn in ihm ein erster Aufruf eines Programmteils eingefügt wird,

Fig. 4 eine prinzipielle Darstellung eines Stack-Speichers, wenn der aufgerufene Programmteil sich bereits im Stack-Speicher befindet,

Fig. 5 eine prinzipielle Darstellung, die angibt, wie die höchste Anzahl von Verdrängungs-Misses mit Hilfe des Stack-Speichers ermittelt wird,

Fig. 6 eine prinzipielle Darstellung der Abbildung der Programmteile auf dem Cache.

Das erfindungsgemäße Verfahren wird zunächst anhand von **Fig. 1** und **Fig. 2** erläutert.

Ausgegangen wird von einem Programm PR, bei dem die einzelnen Programmteile PT entsprechend **Fig. 1a** gebunden sind (Reihenfolge RF0). Es sind dabei Programmteile PT1 bis PTn als Beispiel angegeben. Die Programmteile können z. B. Prozeduren oder Funktionen oder sonstige Programmodule sein.

In einem ersten Schritt des Verfahrens werden nun pseudozufällig die Reihenfolgen der Programmteile geändert, so daß neue Reihenfolgen des gebundenen Programms entstehen. Dadurch können z. B. k (beliebige ganze Zahl) neue Reihenfolgen der Programmteile entstehen. In **Fig. 1b** sind zwei solcher neuen Reihenfolgen gezeigt, bei der die erste (RF1) durch eine erste Vertauschung VT1 und die zweite (RFk) durch eine k-te Vertauschung VTK erzeugt worden ist.

Die durch Vertauschung der Reihenfolge der Programmteile entstandenen neuen Reihenfolgen des Programms gemäß **Fig. 1b** werden nun daraufhin bewertet, mit welcher Laufzeit sie beim Programmlauf ablaufen. Die dabei gefundenen Bewertungsergebnisse der Reihenfolge RF werden miteinander verglichen und dann die Reihenfolge von Programmteilen ausgewählt, deren Laufzeit minimal ist oder dessen Bewertungsergebnis am besten ist.

Anschließend wird von dieser ausgewählten Reihenfolge von Programmteilen ausgegangen und nun wiederum pseudozufällig zwei Programmteile vertauscht, wie dies in **Fig. 2** beispielhaft gezeigt ist. In **Fig. 2** werden z. B. die Programmteile PT4 und PTn miteinander vertauscht. Dies können auch Programmteile sein, die nicht benachbart zueinander liegen. Die dadurch entstehende neue Reihenfolge der Programmteile wird nun wiederum bewertet bezüglich der Laufzeit. Ist das Bewertungsergebnis günstiger als die der Ausgangsreihenfolge, so wird diese als aktuelle Reihenfolge weiter verwendet, sonst die Ausgangsreihenfolge beibehalten. Die nun aktuelle Reihenfolge der Programmteile wird wiederum diesem Schritt unterzogen, es werden also wiederum zwei Programmteile der aktuellen Reihenfolge pseudozufällig miteinander vertauscht und wiederum bewertet. Dieses Verfahren wird so lange weitergeführt, bis ein vorgegebenes Abbruchkriterium erfüllt ist. Dieses Abbruchkriterium kann eine vorgegebene Anzahl von Vertauschungsversuchen sein oder eine vorgegebene Anzahl aufeinander folgender, erfolgloser Vertauschungsversuche sein.

Mit dem Verfahren lassen sich gute Konfigurationen der Programmteile-Reihenfolgen finden, deren Bewertungsergebnis höchstens geringfügig höher ist als das der optimalen Konfiguration. Um noch Reihenfolgen mit besseren Ergebnissen zu finden, ist auszuschließen, daß sich die Trajektorie im Permutationsraum in einem lokalen Minimum des Bewertungsergebnisses verfängt, also bei Bewertung z. B. mit Hilfe der Kostenfunktion sich bei einem lokalen Minimum der Kosten verfängt. Um dies zu vermeiden, müssen Programmteilvertauschungen mit einer gewissen Wahrscheinlichkeit auch dann zugelassen werden, wenn sich höhere Kosten bei der Bewertung ergeben. Deshalb kann das Verfahren durch einen weiteren Schritt ergänzt werden, und zwar durch eine Bedingung, die neue Reihenfolge auch dann bei den weiteren Programmteil-Vertauschungen als aktuelle Reihenfolge zunehmen, wenn die Kostenverschlechterung kleiner ist als eine Zahl, die ein Pseudozufallszahlengenerator generiert hat (normiert auf die Höhe der Kosten). Alternativ dazu kann auch die Bedingung p exp (-Kostenverschlechterung) mit einer Pseudozufallszahl p ($0 \leq p < 1$) verwendet werden, die zu große Verschlechterungen in den Kosten vermeidet.

Zur Bewertung der erzeugten Reihenfolgen der Programmteile bezüglich der Laufzeit des Programms kann ein bekannter Cache-Simulator verwendet werden, wie er sich z. B. aus der Literaturstelle Alan Jay Smith, "Cache Memories", ACM Computing Surveys, Vol. 14, No. 3, Sept. 1982, pp. 473—530, James R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic", Proc. of the 10th Annual Symposium on Computer Architecture, June 1983, pp. 124—131, Mark Hill and Alan Jay Smith, "Experimental Evaluation of On-Chip Microprocessor Cache Memories", Proceedings of the 11th Annual Symposium on Computer Architecture, June 1984, pp. 158—166 und James E. Smith and James R. Goodman, "A Study of Instruction Cache Organizations and Replacement Policies", Proceedings of the 10th Annual Symposium on Computer Architecture, June 1983, pp. 132—137 ergibt. Es ist aber auch möglich, mit Hilfe einer Kostenfunktion die Bewertung durchzuführen. Wie dies erfolgt, wird anschließend beschrieben.

Der erste Schritt des Kostenfunktionsverfahrens besteht darin, mit einem Trace-Verfahren das vorgegebene Programm zu untersuchen, um festzustellen, wann bei Ablauf des Programms Programmteile aufgerufen werden. Dies kann z. B. mit dem folgenden Algorithmus 1, der im Pseudocode dargestellt ist, verdeutlicht werden, wobei hier als Programmteile Prozeduren verwendet werden. Derartige Trace-Verfahren sind bekannt und brauchen nicht ausführlich erläutert zu werden.

Algorithmus 1
Eingabeparameter: keine
Ausgabeparameter: keine
solange (letztes Trace-Element nicht erreicht)
hole Trace-Element
bestimme Prozedurdeskriptor zu Trace-Element
falls (Trace-Element = Prozeduraufruf) erhöhe Prozeduraufrufzähler
erhöhe Prozedur-Instruktionszähler
falls (Trace-Element innerhalb Schleife)
solange (Trace-Element innerhalb Schleife)
bestimme Schleifendeskriptor zu Trace-Element
falls (Trace-Element = Prozeduraufruf) erhöhe Prozeduraufrufzähler für Schleife
erhöhe Schleifen-Instruktionszähler
falls (Trace-Element = Schleifenende) erhöhe Schleifen-Iterationszähler
bestimme nächste Schleife.

Ein Trace-Element des Algorithmus 1 beschreibt die Verwendung eines Codebereiches, z. B. einer Prozedur, Schleife, Basisblock oder Instruktion. Die einzelnen Trace-Elemente im Beispiel werden nun daraufhin untersucht, ob sie einen Prozeduraufruf darstellen und ob eine Schleife vorliegt. Mit Hilfe der Prozedurdeskriptoren wird ein Trace-Element der zugehörigen Prozedur zugeordnet, mit Schleifendeskriptoren wird geprüft, ob das Trace-Element von Schleifen umgeben ist. Im übrigen ist der Algorithmus 1 aus sich heraus verständlich.

Das Ergebnis des Traceverfahrens wird ausgewertet, um festzustellen, ob dynamische Konfliktmengen erzeugt worden sind. Eine Auswertung erfolgt im Hinblick auf die Prozeduraufrufe und Prozedurrücksprünge des Programms. Demgemäß kann das Ergebnis des Traceverfahrens z. B. folgendermaßen dargestellt werden.

0 1..i..j..i 1 0

Dabei repräsentieren i, j Prozedurnummern, $0 \leq i \leq j \leq n_p$, wobei Prozedur 0 stets die Startprozedur sei und n_p die Prozedur mit der großen Prozedurnummer.

Anhand eines Beispieles sei dies weiter erläutert: Die Prozedurfolge sei z. B.

(0 1 2 1 2 1 0 3 0)

Die Startprozedur 0 eines Programms ruft die Prozedur 1 auf, die Prozedur 1 ruft wiederum die Prozedur 2 auf. Wenn die Prozedur 2 abgearbeitet ist, findet ein Rücksprung zur aufrufenden Prozedur statt. Prozedur 1 ruft jetzt Prozedur 2 nochmals auf. Danach erfolgt erneut der Rücksprung zur Prozedur 1. Terminiert Prozedur 1, findet ein Rücksprung zu Prozedur 0 statt. Prozedur 0 ruft schließlich Prozedur 3 auf und das Programm ist beendet mit der Startprozedur 0.

Lokalitätsinformation ist im Trace-Ergebnis in Form von zeitlicher Aufrufhäufigkeit und Aufrufreihenfolge der Prozeduren enthalten. Dieses Trace-Ergebnis wird jetzt untersucht, ob Konfliktmengen vorliegen, die zu einem Verdrängungs-Miss führen können.

Eine Konfliktmenge x_1, x_2 liegt dann vor, wenn sich aus dem zeitlichen Ablauf des Traceverfahrens die Möglichkeit ergibt, daß eine im Cache befindliche Prozedur x_1 durch eine Prozedur x_2 verdrängt werden kann und umgekehrt. Betrachtet wird dazu eine Prozedur x zu einem Zeitpunkt t_2 und deren zeitlich letzte Verwendung zum Zeitpunkt t_1. Alle zwischen den Zeitpunkten t_1 und t_2 aufgerufenen Prozeduren sind Konfliktpartner zur Prozedur x. Das betrachtete Zeitintervall (t_1, t_2), im weiteren Lokalitätsintervall genannt, enthält die Prozedur x genau zweimal, nämlich am Anfang t_1 und am Ende des Intervalls t_2. Das Lokalitätsintervall (0, 1, 2, 1, 2, 1, 0) des Traceergebnisses aus obigem Beispiel enthält die Prozedur 0 am Anfang und Ende des Lokalitätsintervalls. Wie die Prozeduren aus dem Cache abgebildet werden, bzw. welche Prozeduren sich tatsächlich im Cache verdrängen, ist aus diesen Daten nicht ersichtlich. Wichtig ist aber, daß alle Prozeduren innerhalb des Lokalitätsintervalls potentiell mit der Prozedur x in Konflikt stehen können. Folgende Konfliktmengen lassen sich mit dem Lokalitätsintervall (0 1 2 1 2 1 0) bilden:

(0,1): Prozedur 0 und 1 bilden eine Konfliktmenge
(0,2): Prozedur 0 und 2 bilden eine Konfliktmenge
(0, 1, 2): Prozedur 0, 1 und 2 bilden eine Konfliktmenge

Mit obigem Beispiel erhält man folgende Lokalitätsintervalle und Konfliktmengen (angeordnet nach dem zeitlichen Auftreten der Lokalitätsintervalle):

	Lokalitätsintervall	Konfliktmengen
(0 1 2 1 2 1 0 3 0):	(0, 3, 0)	(0, 3)
(0 1 2 1 2 1 0 3 0):	(0, 1, 2, 1, 2, 1, 0)	(0, 1), (0, 2), (0, 1, 2)
(0 1 2 1 2 1 0 3 0):	(1, 2, 1)	(1, 2)
(0 1 2 1 2 1 0 3 0):	(2, 1, 2)	(1, 2)
(0 1 2 1 2 1 0 3 0):	(1, 2, 1)	(1, 2)

Die Konfliktmengen kann man Gewichten nach der Anzahl ihrer korrespondierenden Lokalitätsintervalle im Prozedurtrace-Ergebnis.

In unserem Beispiel würden somit die Konfliktmengen folgendermaßen gewichtet:

(0,1): gew((0,1)) = 1
(0,2): gew((0,2)) = 1
(0,3): gew((0,3)) = 1
(1,2): gew((1,2)) = 3
(0,1,2): gew((0,1,2)) = 1

Wird die Gewichtung der mehrelementigen Konfliktmengen genauer betrachtet, so ist erkennbar, daß deren Gewichtung bereits in die Gewichtung der Untermengen eingegangen ist. Wird eine i-elementige Konfliktmenge gewichtet (i > 2), so werden auch alle 2-bis (i − 1)-elementigen Untermengen der Konfliktmengen gewichtet. Im obigen Beispiel ist die Gewichtung der Konfliktmenge (0, 1, 2) in der Gewichtung der Konfliktmengen (0, 1), (0, 2) und (1, 2) enthalten. In einem ersten Ansatz genügt es also, für i-elementige Konfliktmengen nur 2-elementige Konfliktmengen zu betrachten.

Die Bildung und Gewichtung der Konfliktmengen aus Lokalitätsintervallen läßt sich algorithmisch mit einem LRU-Stack-Speicher realisieren. Dies wird mit Hilfe der **Fig.** 3 bis 4 erläutert. Ein LRU-Stack $S_n(t)$: $=(x_1 \ldots x_n)$ ist ein Stackspeicher LRU der Tiefe n zur Aufnahme von maximal n-Stackelementen oder Prozedurnummern zum Zeitpunkt t. Nach **Fig.** 1 enthält der Stackspeicher LRU $(x_1 \ldots x_n)$ Stackelemente, wobei X_1 das oberste Stackelement darstellt, das zuletzt referenziert oder aufgerufen worden ist. Wird ein Trace-Element x_i bzw. die Prozedur x_i aufgerufen, so wird sie auf den Stack gelegt. Unterscheidet sich eine Prozedurnummer x_i von den bisherigen im Stack enthaltenen Nummern, so wird sie gemäß **Fig.** 3b in den Stack eingefügt, d. h. an erster Stelle des Stackspeichers LRU wird x_i eingefügt, an letzter Stelle das Element x_n gelöscht. Alle übrigen Stackelemente werden um eine Position verschoben.

Wird jedoch eine Prozedur aufgerufen, die bereits im Stackspeicher LRU enthalten ist, dann wird gemäß **Fig.** 4 vorgegangen. Die Prozedurnummer sei x_i, siw wird gemäß **Fig.** 4b an die oberste Stelle des Stackspeichers LRU eingefügt, die übrigen Stackelemente bis zum bisherigen Stackelement $x_i = x_k$ um eine Stelle verschoben. Somit werden die Elemente eines Lokalitätsintervalls mit dem LRU-Stackspeicher identifiziert, wenn man alle Elemente des Stackspeichers mit der Prozedur x_i zum Zeitpunkt t des Prozedurtraces vergleicht. Ist die Prozedur x_i zum Zeitpunkt t im LRU-Stackspeicher, so sind die Stackelemente $x_1, \ldots x_i - 1$ Elemente des korrespondierenden Lokalitätsintervalls bzw. bilden Konfliktpartner zur Prozedur x_i.

Mit Hilfe der **Fig.** 5 wird dieses Verfahren für das angegebene Beispiel gezeigt. Es wird dabei davon ausgegangen, daß der Stackspeicher LRU n = 3 Elemente aufnehmen kann. Dies ist ausreichend, da in dem Beispiel nur drei verschiedene Prozeduren außer der Startprozedur vorhanden sind.

Fig. 5 zeigt nun für das Beispiel zu verschiedenen Zeitpunkten den Inhalt des Stackspeichers LRU. Zu dem Zustand des Stackspeichers ist angegeben, welche Konfliktmengen sich ergeben und wie diese Konfliktmengen zu gewichten sind. Zunächst wird die Prozedur 0 in den Stackspeicher geschoben, anschließend die Prozedur 1 gefolgt von der Prozedur 2. Wenn jetzt wiederum die Prozedur 1 aufgerufen wird, wie dies im Beispiel der Fall ist, dann wird eine erste Konfliktmenge, gebildet aus den Prozedurnummern 1 und 2, festgestellt. Dementsprechend wird in der zugeordneten Spalte die Reihenfolge der Prozeduren 1, 2 mit 1 gewichtet. Anhand der **Fig.** 5 kann das weitere Vorgehen leicht erkannt werden. Auf diese Weise lassen sich die Konfliktmengen der Prozeduren in den Lokalitätsintervallen feststellen und auch eine Gewichtung durchführen.

Es ist zu sehen, daß die Feststellung einer Konfliktmenge immer erfordert, daß eine Prozedur mindestens zweimal referenziert, also adressiert worden ist. Beim ersten Aufruf wird die Prozedur nur in den LRU-Stackspeicher aufgenommen. Beim zweiten Aufruf wird dann die Konfliktmenge festgestellt und gewichtet.

Ein Algorithmus zum Bilden und Gewichten der Konfliktmengen könnte wie Algorithmus 2 aussehen:

Algorithmus 2
Eingabeparameter: keine
Ausgabeparameter: keine
solange (letztes Trace-Element nicht erreicht)
hole Trace-Element
Stackelement$_1$:= top_ 1rustack()
solange (Stackelement$_1$ # letztes Stackelement)
falls (Trace-Element = Stack-Element$_1$)
Stack-Element$_2$:= top_ 1rustack()
solange (Stack-Element$_1$ # Stack-Element$_2$)
valuate conflictset (Trace-Element, Stack-Element$_2$)
hole nächstes Stack-Element$_2$
sonst hole nächstes Stack-Element$_1$

Der Algorithmus ist wiederum im Pseudocode geschrieben und damit leicht verständlich. Es wird das Stackelement 1 und das Stackelement 2 unterschieden. Das Stackelement 1 entspricht dem Element des Traceergebnis, das mit den im Stackspeicher enthaltenen Stackelementen 2 verglichen wird. Solange das Stackelement 1 ungleich ist zum Stackelement 2, wird die Nummer des Stackelementes 2 notiert und ebenso das Traceelement oder das Stackelement 1.

Nachdem die Konfliktmengen festgestellt worden sind und zudem die Gewichtung durchgeführt worden ist, wird für jedes aus dem Traceverfahren ermittelte Paar untersucht, wie groß gemessen in Cacheblöcken die Überlappung der beteiligten Programmteile ist. Dazu ist die Cachegröße vorgegeben. Der Verfahrensschritt kann anhand der **Fig.** 6 nachvollzogen werden. Es wird davon ausgegangen, daß die Anzahl der maximal möglichen Konflikte (siehe A), d. h. die gewichteten Konfliktmengen zwischen der Prozedur 1 und der Prozedur 3 gleich 4 und der Prozedur 2 und Prozedur 4 gleich 6 ist. Unter der Annahme, daß die Prozeduren entsprechend der Figur aufeinanderfolgen und unter der Annahme, daß der Adreßraum ADR des Caches in Cacheblöcke CB unterteilt ist, ergibt sich die Abbildung des Programms aus den Prozeduren 1 bis 4 aus der Darstellung nach **Fig.** 6. Die Prozedur 1 nimmt drei Cacheblöcke ein, die Prozedur 2 vier Cacheblöcke, die Prozedur 3 sechs Cacheblöcke und die Prozedur 4 zwei Cacheblöcke. Wenn der Adreßraum des Cache neun Cacheblöcke enthält, dann kann die Prozedur 1 und Prozedur 2 ganz im Cache stehen, die Prozedur 3 nur mit zwei Cacheblöcken und die Prozedur 4 überhaupt nicht. Das heißt, eine Abbildung dieses Programms aus vier Prozeduren auf den Cachespeicher der **Fig.** 4 würde ergeben, daß der Anzahl der kollidierenden Cacheblöcke bei Prozedur 1 und 3 gleich 3 ist, bei Prozedur 2 und Prozedur 3 gleich 1 und bei Prozedur 2 und Prozedur 4 gleich 2 ist (siehe B). Die Untersuchung des Programms auf mögliche Konfliktmengen hat ergeben, daß nur zwischen der Prozedur 1 und der Prozedur 3 bzw. der Prozedur 2 und der Prozedur 4 eine Konfliktmenge besteht. Die mögliche Anzahl von Verdrängungs-Misses kann sodann nach folgender Formel berechnet werden:
Summe über aller Konfliktpaare nach A multipliziert mit der Anzahl gemeinsam genutzter Cacheblöcke nach B ergibt im Ausführungsbeispiel der **Fig.** 4 folgenden Wert:

$$3 \times (\text{Prozedur 1, Prozedur 3}) + 2 \times (\text{Prozedur 2, Prozedur 4}) = 24$$

Bei einer festgelegten Folge der Prozeduren eines Programmes gemäß **Fig.** 4 können somit maximal 24 Verdrängungs-Misses auftreten. Für eine andere Folge der Prozeduren des Programms würde sich dieser Wert ändern. Somit wird mit dem Verfahren ein Weg gezeigt, wie bei einer Reihenfolge von Programmteilen, z. B. Prozeduren, die Anzahl der höchsten möglichen Verdrängungs-Misses feststellbar ist.

Patentansprüche

1. Verfahren zum Binden von Programmteilen zu einem Programm, das von einem nach dem Cache-Prinzip arbeitenden Rechner mit minimaler Laufzeit abgearbeitet werden kann,
   a) bei dem pseudozufällig eine Anzahl von Reihenfolgen der Programmteile erzeugt worden und diese Anzahl der Reihenfolgen bezüglich der Laufzeit bewertet werden und eine erste Reihenfolge von Programmteilen ausgewählt wird, die das beste Bewertungsergebnis, also die geringste Laufzeit hat,
   b) bei dem mit der ersten Reihenfolge der Programmteile weitergearbeitet wird und pseudozufällig jeweils zwei Programmteile vertauscht werden und dann festgestellt wird, ob die geänderte Reihenfolge zu einem besseren Bewertungsergebnis führt und für diesen Fall mit der geänderten Reihenfolge als aktuelle Reihenfolge weitergearbeitet wird, sonst mit der nicht geänderten Reihenfolge der Programmteile,
   c) und bei dem der Schritt b) so lange durchgeführt wird, bis ein vorgegebenes Abbruchkriterium erfüllt ist.
2. Verfahren nach Anspruch 1, bei dem trotz negativem Ergebnis der Bewertung mit der geänderten Reihenfolge der Programmteile weitergearbeitet wird, wenn die Verschlechterung des Bewertungsergebnisses kleiner ist als eine von einem Pseudozufallszahlengenerator erzeugte Zufallszahl, die auf das Bewertungsergebnis der aktuellen Reihenfolge normiert ist.
3. Verfahren nach einem der Ansprüche 1 oder 2, bei dem das Bewertungsverfahren dann beendet wird, wenn eine vorgegebene Anzahl von Bewertungen durchgeführt worden ist.
4. Verfahren nach einem der vorhergehenden Ansprüche, bei dem die Bewertung mit Hilfe eines Cache-Simulators durchgeführt wird.

5. Verfahren nach einem der Ansprüche 1 bis 3, bei dem die Bewertung auf folgende Weise durchgeführt wird,

a) ein Trace-Verfahren für das Programm aus den Programmteilen wird durchgeführt, das als Trace-Ergebnis Adreßaufrufe zu Codebereichen oder Datenbereichen der Programmteile enthält,

b) es wird aus dem Trace-Ergebnis festgestellt, wie oft sich zwei Programmteile höchstens verdrängen können,

c) es wird für jeweils ein Paar von Programmteilen festgestellt, wie groß in Cache-Blöcken ausgedrückt und bei Abbildung des Programms auf den Cache die Überlappung dieses Programmteilepaares ist,

d) es wird für jedes Paar von Programmteilen das Produkt aus der Anzahl der Verdrängungen und der Anzahl der kollidierenden sich überlappenden Cache-Blöcke gebildet,

e) es wird zur Ermittlung der Anzahl der Verdrängungs-Misses des Programms und damit als Bewertungsergebnis die Summe über alle Produkte gebildet.

6. Verfahren nach Anspruch 5, bei dem zur Bestimmung der Anzahl der Verdrängungen der Programmteilepaare anhand der Aufruffolge der Programmteile festgestellt wird, welche Programmteile zwischen zwei aufeinanderfolgenden Aufrufen eines Programmteiles liegen und daraus ermittelt wird, wie oft die Programmteile eines Paares nacheinander aufgerufen werden.

7. Verfahren nach Anspruch 6, bei dem zur Bestimmung der Anzahl der Verdrängungen ein LRU-Stack-Speicher verwendet wird, in dem die die aufrufenden Programmteile kennzeichnenden Elemente hineingeschoben werden, bei dem bei Aufruf eines ersten Programmteiles aus dem Trace-Ergebnis durch Vergleich dieses ersten Programmteiles mit bereits im LRU-Stackspeicher enthaltenen Programmteil von oben beginnend festgestellt wird, ob dieser erste Programmteil schon einmal aufgerufen worden ist und für diesen Fall die oberhalb des im LRU-Stackspeicher enthaltenen ersten Programmteiles liegenden Programmteile notiert werden und anschließend der erste Programmteil an die erste Stelle des LRU-Stackspeichers geschoben wird.

Hierzu 4 Seite(n) Zeichnungen

– Leerseite –

Nummer: DE 42 06 567 C1
Int. Cl.[5]: G 06 F 9/45
Veröffentlichungstag: 27. Mai 1993

FIG 3

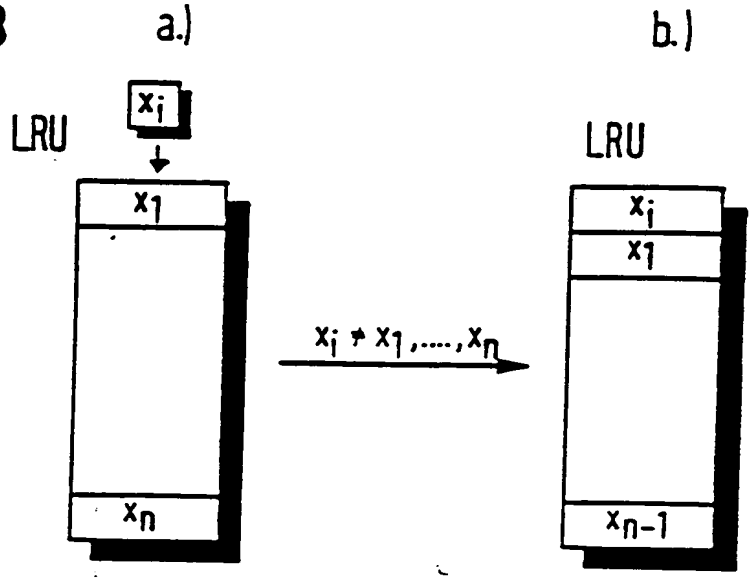
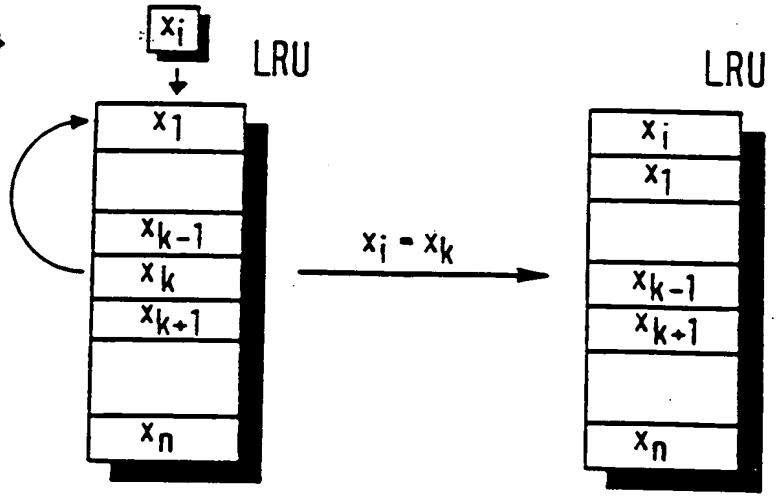
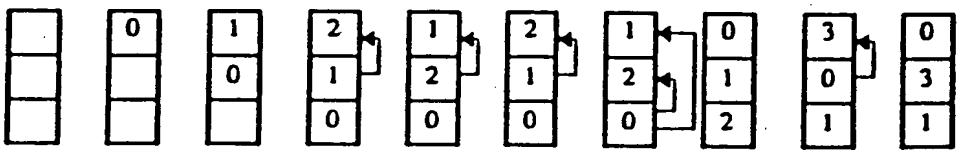


FIG 4



Nummer: DE 42 06 567 C1
Int. Cl.[5]: G 06 F 9/45
Veröffentlichungstag: 27. Mai 1993

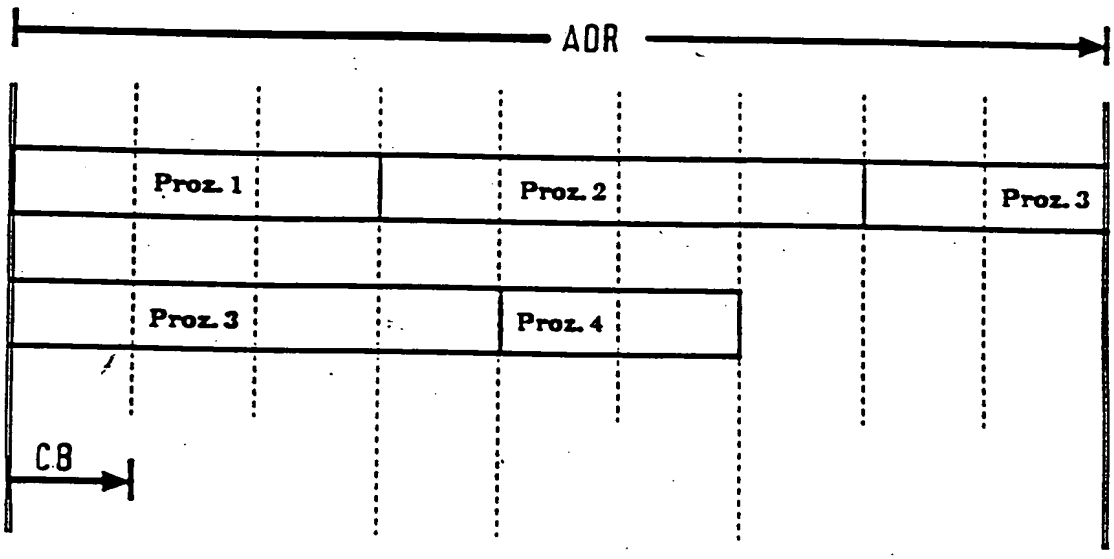
FIG 5

Prozedur - Trace	⟨0	1	2	1	2	1	0	3	0⟩	
LRU-Stack (n=3)										
Gebildete Konflikt-mengen				{1,2}	{1,2}	{1,2}	{0,1} {0,2}		{0,3}	
gew ({0,1})	0	0	0	0	0	0	1	1	1	1
gew ({0,2})	0	0	0	0	0	0	1	1	1	1
gew ({0,3})	0	0	0	0	0	0	0	0	1	1
gew ({1,2})	0	0	0	1	2	3	3	3	3	3

Nummer: DE 42 06 567 C1
Int. Cl.[5]: G 06 F 9/45
Veröffentlichungstag: 27. Mai 1993

FIG 6

Anzahl maximal möglicher Konflikte zwischen [Prozedur X und Prozedur Y] z.B.
[Proz. 1,Proz. 3] = 4
[Proz. 2,Proz. 4] = 6 } A



Anzahl der Cache-Blöcke, auf die sowohl (Prozedur X und Prozedur Y) abgebildet werden
(Proz. 1,Proz. 3) = 3
(Proz. 2,Proz. 3) = 1
(Proz. 2,Proz. 4) = 2 } B

Nummer: DE 42 06 567 C1
Int. Cl.[5]: G 06 F 9/45
Veröffentlichungstag: 27. Mai 1993

FIG 1

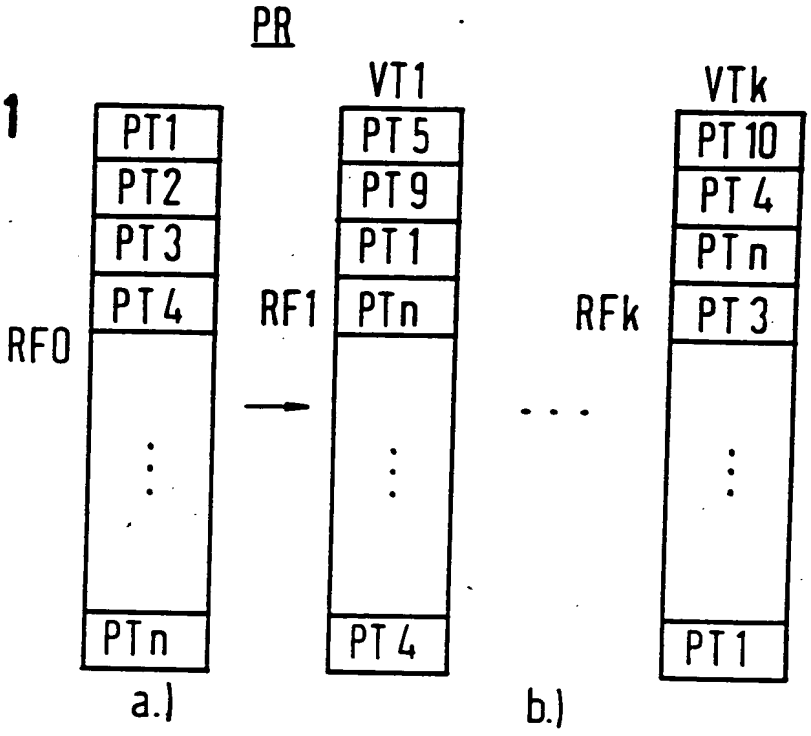


FIG 2